컴파일러 과제2
보고서

20181755 이건희

2024년 9월

1. 개발 환경

과제1의 환경과 동일한 wsl2 상의 Ubuntu 22 LTS, gcc 11, gdb 12이다.

2. 코드 설명

사실 해결방안이 너무나 당연하기에 대부분의 학생이 같거나 비슷한 방법으로 과제를 해결했으리라 생각한다.

1) '-'(이하 MINUS), '/'(이하 SLASH)를 yylex()의 분기에 추가
2) expression()에서 MINUS, term()에서 SLASH 연산 추가
3) 실수 지원을 위해 union 사용, yylex()의 숫자 파싱 부분에서 소숫점 탐색
4) 에러 처리의 심화

보고서의 내용이 다들 너무나 뻔할 것이기에, 과제를 해결하면서 스스로 좀 기발하게 해결했다 생각한 부분들을 위주로 서술하도록 하겠다.

2-1. 정수가 아닌 실수(Real Number)의 처리

본 과제는 실수의 처리를 요구한다. BNF로 작성된 사칙연산의 규칙은 모두 lvalue, rvalue의 연산으로 정해져 있다. 이를 처음에는 (정수||실수)^2 = 4가지의 경우의 수를 나누어서 경우에 따라 필요한 값만 캐스팅 해주는 방식으로 처리하도록 만들었으나, 코드가 너무 지저분해져서 다른 방법을 생각해보았다.

결과적으로 모든 연산에서 모든 수를 float으로 처리하되, 이의 결과가 정수가 되면(ex. 10.00f) 정수로 다시 변환하는 식으로 만들었다.

```
void fconv(val lval, val rval, float *tmp1, float *tmp2) {
    if (lval.t == INT) {
        *tmp1 = (float) lval.x.i;
    }
    else {
        *tmp1 = lval.x.f;
    }

    if (rval.t == INT) {
        *tmp2 = (float) rval.x.i;
    }
    else {
        *tmp2 = rval.x.f;
    }
```

fconv 함수는 연산을 위해 필요한 tmp1, tmp2에 lvalue, rvalue가 정수일 경우 float으로 캐스팅하여 저장, 실수일 경우 단순 복사를 해주는 함수이다.

```
float fcalc(float tmp1, float tmp2, token_type op) {
    switch (op) {
        case PLUS: {
            return tmp1 + tmp2;
        }
        case MINUS: {
            return tmp1 - tmp2;
        }
        case STAR: {
            return tmp1 * tmp2;
        }
        case SLASH: {
            return tmp1 / tmp2;
        }
        default: {
            return 0.0f;
        }
    }
}
```

이 덕분에 expression(), term()의 네 operator의 연산마저 fcalc()라는 하나의 단순한 함수로 묶을 수 있었다.

```
val fcani(val r) {
    val ret = VALDFLT;

#if DEBUG
    ...
#endif

    if (fmod( x: r.x.f,  y: 1.0) == 0.0) {
        ret.t = INT;
        ret.x.i = (int) r.x.f;
    }
    else {
        ret.t = FLOAT;
        ret.x.f = r.x.f;
    }

    ret.e = r.e;

#if DEBUG
    ...
#endif

    return ret;
}
```

fcani()는 float값이 정수로 변환 가능할 경우 바꿔주는 함수이다. 예를 들어 “2.5*4”, “4.5+5.5”와 같은 수식이 주어진다면, 이의 fcalc()의 결과는 10.00f이다. 하지만 fmod(floating modulus)를 사용하여 1.0으로 나누었을때의 나머지가 0이라면, 이는 정수라 판단이 가능하다. fmod()의 결과가 참일 경우 정수로 변환, 거짓일 경우 그대로 값을 유지해준다.

```
val expression() {

#if DEBUG
    ...
#endif

    val lval = VALDFLT;
    lval = term();
    token_type op = token;

    EXITi (errstatus);

    while (token == PLUS || token == MINUS) {
        yylex();
        val rval = term();

        float tmp1 = 0.0f;
        float tmp2 = 0.0f;

        fconv(lval, rval, &tmp1, &tmp2);

        val rslt = VALDFLT;
        rslt.t = FLOAT;
        rslt.x.f = fcalc(tmp1, tmp2, op);

        // 정수로 변환이 가능할 경우 변환해줌
        lval = fcani( r: rslt);
    }
```

```
val term() {

#if DEBUG
    ...
#endif

    val lval = VALDFLT;
    lval = factor();
    token_type op = token;

    EXITi(errstatus);

    while (token == STAR || token == SLASH) {
        yylex();
        val rval = factor();

        float tmp1 = 0.0f;
        float tmp2 = 0.0f;

        fconv(lval, rval, &tmp1, &tmp2);

        /** dif by zero 에러 처리 */
        if (op == SLASH) {
            if ((rval.t == INT && rval.x.i == 0) ||
                (rval.t == FLOAT && rval.x.f == 0.0f)) {
                errstatus = -4;  // div by zero
                RETERR;
            }
        }

        val rslt = VALDFLT;
        rslt.t = FLOAT;
        rslt.x.f = fcalc(tmp1, tmp2, op);

        // 정수로 변환이 가능할 경우 변환해줌
        lval = fcani( r: rslt);
```

이 fconv(), fcalc(), fcani() 덕분에 lvalue와 rvalue의 수의 종류에 따른 4가지의 if-else 분기를 거치지 않아 코드가 상당히 깔끔하게 완성되었다. 다만 모든 연산의 과정에 캐스팅을 하는 것에 대한 연산량의 증가는 단점이 될 수 있다.

2-2. MINUS로 시작하는 수식

처음 만든 코드에서 “1-2”는 연산이 되었으나, “-2+1”은 연산이 되지 않았다. recursive-descent parser는 파싱을 시작하기 전 토큰 하나를 읽고 가고, 이것이 NUMBER이어야 한다. 본 과제의 코드에서는 calc()(강의교안 상으로는 main()에 해당)에서 이것이 이루어지는데, 스트링의 첫 문자인 ‘-’가 파싱되어 시작 토큰이 NUMBER가 아닌 MINUS가 되어버린다.

이는 아주 간단한 방법으로 해결이 가능했다. 계산해야 할 수식 스트링의 앞에 단순히 “0”만 덧붙히는 것이다. 위의 예시에서 “1-2”는 “01-2”가 되는데, “01”은 yylex_num()(yylex()에서 수를 파싱하는 부분이 너무 길어져서 다른 함수로 분리함)에서 어차피 1로 파싱이 될 것이고, “-2+1”은 “0-2+1”이 되는데, 이의 결과값은 원래의 수식 스트링과 다름이 없다.

```
155         /** 3. 동적할당 한 메모리에 수식 저장 */
156         int count = 0;
157
158         while (1) {
159             char buf[MAX_LINE_LEN];
160             memset( s: buf,  c: 0,  n: sizeof(char) * MAX_LINE_LEN);
161             buf[0] = '0';
162
163             char *readline = fgets( s: buf + 1,  n: MAX_LINE_LEN - 1,  stream: fp);
164             buf[strcspn( s: buf,  reject: "\n\r")] = 0;
165             if (readline == NULL) {
166                 break;
167             }
168
169             int lidx = 0;
170             for(int i=0; i<strlen( s: buf); i++){
171                 if(buf[i] != ' '){
172                     lines[count][lidx++] = buf[i];
173                 }
174             }
175
176             count++;
177         }
```

사진의 main()에서의 while 루프는 파일에서 한 줄씩 읽어서 -> 버퍼에 저장하고 -> list[x]에 공백을 제외한 char를 복사한다. buf[0]에 ‘0’을 저장하고 fgets는 buf[1]부터 저장하도록 수정했기에, “0~” 형태의 수식이 lists[x]에 저장될 것이다.

```
184         /** 5. 값 출력 */
185         int digit = countdigits( number: count);
186
187         float precision = powf( x: 10,  y: PRECISION);
188
189         for (int i = 0; i < count; i++) {
190             printf( format: "%0*d. %s = ", digit, i + 1, lines[i] + 1);
191
192             if (answers[i].e) { // 에러 발생했을 때
193                 printerr( ecode: answers[i].e);
194             }
195             else {
196                 if (answers[i].t == INT) {
197                     printf( format: "%d\n", answers[i].x.i);
198                 }
199                 else {
200                     printf( format: "%.*f\n", PRECISION,
201                             (roundf( x: answers[i].x.f * precision) / precision));
202                 }
203             }
204         }
```

정답 출력 중 원래 수식을 출력할 때도 조작된 0을 출력하지 않도록 하였다.

2-3. 출력의 정밀도

컴퓨터에서 실수는 IEEE 754에서 정해진 규칙대로 다룬다. 이에 의하면 float(단정밀도 32비트)은 부호부 1비트, 지수부 8비트, 가수부 23비트로 이루어져 있다. 연산에서의 모든 수를 float으로 다루는 것은 이상적이나, 정답을 "12.300000"과 같이 출력하는 것은 의미가 없다.

이를 해결하기 위해 정밀도의 개념을 추가했다.

```c
/******************
 * 선언 & 정의 *
 ******************/
#define MAX_LINE 1000 // 최대 수식의 갯수, 임의로 정함
#define MAX_LINE_LEN 256 // 수식 최대의 길이, 임의로 정함
#define PRECISION 3 // float의 정밀도, 소숫점 n번째 자리까지, 임의로 정함
```

과제의 코드는 PRECISION을 임의로 3으로 선언하였다.

```c
    /** 5. 값 출력 */
    int digit = countdigits( number: count);

    float precision = powf( x: 10, y: PRECISION);

    for (int i = 0; i < count; i++) {
        printf( format: "%0*d. %s = ", digit, i + 1, lines[i] + 1);

        if (answers[i].e) { // 에러 발생했을 때
            printerr( ecode: answers[i].e);
        }
        else {
            if (answers[i].t == INT) {
                printf( format: "%d\n", answers[i].x.i);
            }
            else {
                printf( format: "%.*f\n", PRECISION,
                        (roundf( x: answers[i].x.f * precision) / precision));
            }
        }

    }
```

변수 precision은 10^PRECISION이다. 어떤 수식의 정답이 정수가 아닌 실수라면, 이에 precision을 곱한 뒤 다시 precision으로 나누면 소수점 세 번째 자리까지의 정밀도가 보장된 출력을 한다. 예를 들자면, 정답이 1.234567, 정밀도가 3일 경우 소수점 셋째 자리까지 정확히 출력을 해야한다. 1.234567 * 10^3은 1234.567(소수점을 오른쪽으로 세 칸 이동)이고, 이를 roundf()에 넣으면 1235가 나온다. 이를 다시 10^3으로 나누면 1.235가 된다. printf 포맷 역시 PRECISION만큼의 소수점 아랫자리를 출력하도록 지정했다.

2-4. 소수점의 처리

소수점을 나타내는 '.'(이하 DOT)이 문법에 맞지 않는 경우는 다음과 같이 두 가지가 있다.

1. "1..2"나 "1.2.3"과 같이 한 NUMBER에서 소숫점이 중복 될 경우
2. ".2"와 같이 정수부가 없는 경우(이는 정하기 나름이나, 본 프로그램에서는 오류로 두기로 한다)
3. "3."와 같이 소수부가 없는 경우(역시나 정하기 나름이나, 본 프로그램에서는 오류로 두기로 한다)

소수점이 문법 상 소수점에 오류가 있는 경우는 factor가 제대로 이루어지지 않은 경우이므로, 에러코드는 -1이다(BNF로 나타낸 규칙에서 F ::= N | (E)).

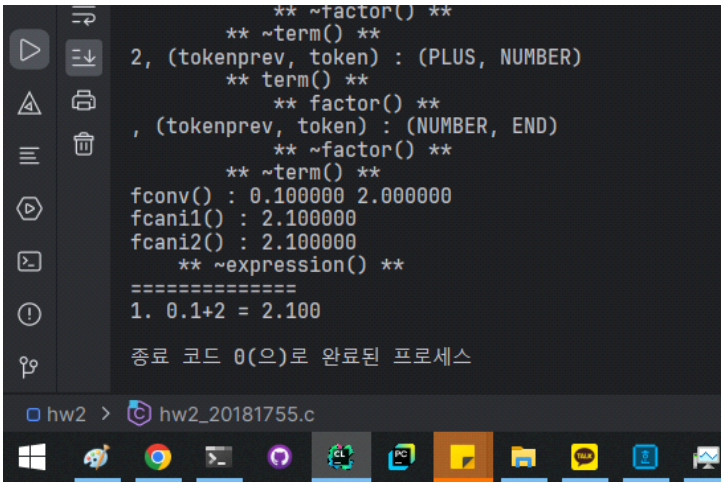
2-4-1. 소숫점이 중복되는 틀린 factor의 처리

```
void yylex_number() {
    memset( s: numbuf,  c: 0,  n: MAX_LINE_LEN);
    numbufprog = 0;

    // 실질적인 숫자 읽기 시작
    numtype = INT;
    bool everdot = false;

    while (isdigit(subptr[0])) {
        numbuf[numbufprog++] = subptr[0];
        subptr++;

        if (subptr[0] == '.') {
            // 정수가 아닌 실수일 때

            // .의 이전이 조작된 0일 때
            if (subptr[-1] == '0' && subptr - 1 == suborigin) {
                token = NOTHING;
                errstatus = -1;
                return;
            }

            if (!everdot) {
                // 한 NUMBER에서 처음 나오는 '.'
                everdot = true;
                numtype = FLOAT;
                numbuf[numbufprog++] = subptr[0]; // '.'도 버퍼로 복사
                subptr++;

                if(!isdigit(subptr[0])){
                    token = NOTHING;
                    errstatus = -1;
                    return;
                }
            }
            else {
                // 한 NUMBER에서 두 번째 나오는 '.'
                token = NOTHING;
                errstatus = -1; // factor가 제대로 되지 않은 상황
                return;
            }
        }
    }
```

yylex_num()은 substring의 시작이 숫자임을 가정하고(isdigit[subptr[0]]) 파싱을 시작한다. while 루프에서 DOT이 처음 나올 경우 numtype = FLOAT, dot을 수의 일부라 생각하고 numbuf에 복사, 수와 마찬가지로 서브스트링의 시작을 한 칸 뒤로 밀어준다. DOT이 두 번째 나올 경우 이는 무조건 에러라 에러코드가 -1로 설정된다.

문제가 되는 것은 2. 의 처리이다. 예를 들어 "1+.2"처럼 두 번째 이후의 NUMBER가 정수부가 없는 float인 수식이 주어질 경우 token의 변화는 NUMBER("1.2"), PLUS("+"), NOTHING("."), 여기까지 진행되어 yylex()에서 에러임이 밝혀지고, 각 함수에서 재귀적으로 EXITi(빠른 에러 처리)가 실행된다.

2-4-2. 정수부가 없는 소수의 처리



"1+.2"와 같이 두 번째 이후의 NUMBER로 정수부가 없는 float이 주어진다면, yylex()의 else 분기를 타고 처리가 된다. 그러나 ".1+2"과 같이 첫 번째 NUMBER가 정수부가 없는 float이 주어진다면 문제가 된다. 2-2에서 첫 토큰으로 MINUS가 나올 경우를 처리하기 위해 입력받은 스트링의 가장 앞에 "0"을 덧붙히기로 했다. 이 때문에 ".1+2"는 "0.1+2"가 되어버리고, 이는 의도치 않은 2.1이라는 결과가 나온다.

이를 해결하기 위해 처음에는 한 NUMBER 안에서 조작된 '0'인지 아닌지 판단하는 boolean 전역변수를 두려 했다. 이 방법으로는 ".1+2"는 오류라 제대로 판단할 수 있지만, "0.1+2"는 "00.1+2"가 되는데, 분명 규칙에 맞는 수식인데도, '0'이 수식의 가장 처음에 나왔다는 이유로 오류라 오판한다. 다시 말해서 "0.n"으로 시작하는 모든 수식이 오류가 있다 판단하는 것이다. 결국 필요한 것은 '0'이 조작된 것인지 아닌지에 대한 판단이다. 이 판단을 어떻게 할지에 대해 조금 생각해보고 진행률을 사용하도록 수정도 해보았으나 작동을 하지 않았다.

더욱 고민이 깊어지다 불현듯 단순히 메모리 주소를 비교한다면 쉽게 판단이 가능하지 않을까 하는 생각이 들었고, 이대로 완성하니 해결이 되었다.

```
65  // 2. 스트링
66  char *suborigin; // substring의 처음 시작 → 조작된 '0'
67  char *subptr; // substring 을 가리키는 포인터
```

```
565 val calc(char *line) {
566     subptr = line; // 처음 시작시 string 전체를 substring으로 생각
567     suborigin = line;
568
569     val result = VALDFLT;
570     errstatus = 0;
571     tokenprev = NOTHING;
572
573     yylex();
574     result = expression();
575
```

메모리 주소로 비교하려면, 원래 스트링의 첫 주소가 필요하다. 이를 전역변수 char *suborigin으로 두었다. 수식의 연산은 calc()로 시작하는데, 매 연산 이전 suborigin을 스트링의 가장 첫 글자의 메모리 주소로 설정해야 한다.

```
void yylex_number() {
    memset( s: numbuf, c: 0, n: MAX_LINE_LEN);
    numbufprog = 0;

    // 실질적인 숫자 읽기 시작
    numtype = INT;
    bool everdot = false;

    while (isdigit(subptr[0])) {
        numbuf[numbufprog++] = subptr[0];
        subptr++;

        if (subptr[0] == '.') {
            // 정수가 아닌 실수일 때

            // .의 이전이 조작된 0일 때
            if (subptr[-1] == '0' && subptr - 1 == suborigin) {
                token = NOTHING;
                errstatus = -1;
                return;
            }

            if (!everdot) {
                // 한 NUMBER에서 처음 나오는 '.'
                everdot = true;
                numtype = FLOAT;
                numbuf[numbufprog++] = subptr[0]; // '.'도 버퍼로 복사
                subptr++;
```

표를 그려 배열을 시각적으로 표현하는게 “.1”이 어떻게 0.1이 되었는지 이해하기 편할 것이다.

	0x10	0x11	0x12	0x13	0x14	0x15
“.1+2” ->	0 (조작)	.	1	+	2	EOF

빨간 박스의 코드가 없다면 흐름은 다음과 같다.

calc()에서 첫 번째 yylex()가 호출이 된다.
yylex_num(){
  while 1번째
    1. 0x10의 ‘0’ numbuf에 복사
    2. subptr은 0x11이 된다.
    if (*0x11 == ‘.’) -> true
      4. everdot = true, numtype = FLOAT
      5. 0x11의 ‘.’ numbuf에 복사
      6. subptr은 0x12가 된다.
  while 2번째 (subptr[0] = *0x12 = ‘1’)
    7. 0x12의 ‘1’ numbuf에 복사
    8. subptr은 0x12가 된다.
  while 3번째는 실행되지 않는다. (subptr[0] = *0x13 = ‘+’)
}
….

이러한 단계를 통해 첫 yylex()후 “.1”이 0.1이 되어버리는 것이다.

```
230  void yylex_number() {
231      memset( s: numbuf,  c: 0,  n: MAX_LINE_LEN);
232      numbufprog = 0;
233
234      // 실질적인 숫자 읽기 시작
235      numtype = INT;
236      bool everdot = false;
237
238      while (isdigit(subptr[0])) {
239          numbuf[numbufprog++] = subptr[0];
240          subptr++;
241
242          if (subptr[0] == '.') {
243              // 정수가 아닌 실수일 때
244
245              // .의 이전이 조작된 0일 때
246              if (subptr[-1] == '0' && subptr - 1 == suborigin) {
247                  token = NOTHING;
248                  errstatus = -1;
249                  return;
250              }
251
252              if (!everdot) {
253                  // 한 NUMBER에서 처음 나오는 '.'
254                  everdot = true;
255                  numtype = FLOAT;
256                  numbuf[numbufprog++] = subptr[0]; // '.'도 버퍼로 복사
257                  subptr++;
```

	0x10	0x11	0x12	0x13	0x14	0x15
“.1+2” ->	0 (조작)	.	1	+	2	EOF

빨간 박스의 코드가 있다면 흐름은 다음과 같다.

```
calc()에서 첫 번째 yylex()가 호출이 된다.
yylex_num(){
  while 1번째
    1. 0x10의 '0' numbuf에 복사
    2. subptr은 0x11이 된다.
    if (*0x11 == '.') -> true
      if (subptr[-1] == '0' && subptr - 1 = origin)
      -> true (&subptr[-1] = 0x10, subptr-1 = 0x10, origin = 0x10)
      에러 처리
}
....
```

빨간 박스의 코드 때문에 첫 NUMBER가 조작된 0임을 알고, 에러 처리할 수가 있다.

2-4-3. 소수부가 없는 소수의 처리

```
256             if (!everdot) {
257                 // 한 NUMBER에서 처음 나오는 '.'
258                 everdot = true;
259                 numtype = FLOAT;
260                 numbuf[numbufprog++] = subptr[0]; // '.'도 버퍼로 복사
261                 subptr++;
262 
263                 if(!isdigit(subptr[0])){
264                     token = NOTHING;
265                     errstatus = -1;
266                     return;
267                 }
268             }
```

“3.+4”와 같이 소수부가 없는 소수가 주어진다면, 이 역시나 에러처리 해줘야 한다. 이는 yylex_num()에서 DOT을 처리해주는 부분에 if문 하나만 추가하여 간단히 해결이 가능하다. 유효한 float은 정수부와 소수부가 전부 있어야 하는 점을 확인하는 원리이다.

yylex_num()이 적당히 진행되어 numbuf에 “3.”까지 저장되어있고, subptr[0]은 ‘+’라 해보자. if문은 다음 while 루프의 조건으로 검사하는 isdigit(subptr[0])을 미리 한다. subptr[0]은 ‘+’이기에 다음 루프 때 조건에 부합하지 않아 빠져나온다는 것을 미리 알 수 있고, 이와 동시에 “3.”이 소수부가 없는 미완성된 float 입력임도 같이 알게 된다.

이러한 방법들로써 소수점으로 인해 발생할 수 있는 에러들을 처리하였다.

2-5. 수식 출력시 가독성 향상

솔직히 그렇게 자랑할만한 내용은 아니긴 하지만 나름 의미가 있다 싶어서 적는다. 이전 과제의 프로그램은 주어진 수식이 예를 들어 "1 + 2- 3"이라 한다면, yylex()에서 빈 칸을 다 날리고, 실질적으로 다루는 char는 1,+,2,-,3,\0(EOF) 이었다.

```
kh@ThinkPad-T16g2:~/compiler/hw1_20181755$ ./hw1 mathproblem1.txt
01. ((1+2))*3 = 9
02. (4+ 5) * 6 = 54
03. 7*8*9 = 504
04. 7 + 4 = 11
05. 1 * 8 = 8
06. 5 + 6*7 = 47
07. 1 * 1 = 1
08. 4 + 3 = 7
09. 3   + 3 + 2 = 8
10. 1 + 3 + 5 = 9
11. 1 +2*3 = 7
12. 1*(2+4) = 6
13. ((2)+(3*(4+5))) = 29
kh@ThinkPad-T16g2:~/compiler/hw1_20181755$
```

그런데 정답을 출력할 때, 문법상만 맞지 보기 싫게 주어진 수식을 그대로 출력하는 점이 거슬렸다.

이를 해결하기 위해 main()의 주어진 수식을 배열에 저장하는 while 루프를 수정했다.

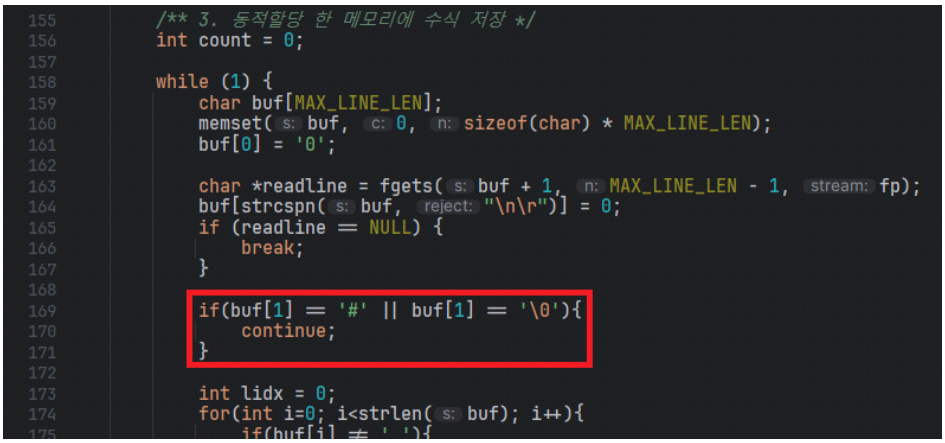
```
154         /** 3. 동적할당 한 메모리에 수식 저장 */
155         int count = 0;
156
157         while (1) {
158             char buf[MAX_LINE_LEN];
159             memset( s: buf,  c: 0,  n: sizeof(char) * MAX_LINE_LEN);
160             buf[0] = '0';
161
162             char *readline = fgets( s: buf + 1,  n: MAX_LINE_LEN - 1,  stream: fp);
163             buf[strcspn( s: buf,  reject: "\n\r")] = 0;
164             if (readline == NULL) {
165                 break;
166             }
167
168             strncpy( dest: lines[count++],  src: buf,  n: strlen( s: buf));
169         }
```

기존의 while 루프 안에서는 버퍼에서 버퍼의 길이만큼 lines[x]에 그대로 strncpy를 했다.

```
155         /** 3. 동적할당 한 메모리에 수식 저장 */
156         int count = 0;
157
158         while (1) {
159             char buf[MAX_LINE_LEN];
160             memset( s: buf,  c: 0,  n: sizeof(char) * MAX_LINE_LEN);
161             buf[0] = '0';
162
163             char *readline = fgets( s: buf + 1,  n: MAX_LINE_LEN - 1,  stream: fp);
164             buf[strcspn( s: buf,  reject: "\n\r")] = 0;
165             if (readline == NULL) {
166                 break;
167             }
168
169             int lidx = 0;
170             for(int i=0; i<strlen( s: buf); i++){
171                 if(buf[i] != ' '){
172                     lines[count][lidx++] = buf[i];
173                 }
174             }
175
176             count++;
177         }
```

수정된 while 루프 안에서는 수식을 일단 버퍼에 저장하되, lines[x]에 복사할 때 ' '(SP)가 아닌 char만 복사하도록 했다. 이로써 무조건 "1-2+3"과 같은 걸러진 스트링만이 lines[x]에 저장될 것이기에, yylex()에서 토큰을 읽기 전 SP를 넘기는 while문은 필요가 없어진다.

2-6. 테스트 파일의 주석기능



```
155     /** 3. 동적할당 한 메모리에 수식 저장 */
156     int count = 0;
157
158     while (1) {
159         char buf[MAX_LINE_LEN];
160         memset( s: buf,  c: 0,  n: sizeof(char) * MAX_LINE_LEN);
161         buf[0] = '0';
162
163         char *readline = fgets( s: buf + 1,  n: MAX_LINE_LEN - 1,  stream: fp);
164         buf[strcspn( s: buf,  reject: "\n\r")] = 0;
165         if (readline == NULL) {
166             break;
167         }
168
169         if(buf[1] == '#' || buf[1] == '\0'){
170             continue;
171         }
172
173         int lidx = 0;
174         for(int i=0; i<strlen( s: buf); i++){
175             if(buf[i] != ' '){
```

테스트 파일에 주석을 못 다니 너무나 불편해서, main()의 빨간 박스 부분을 수정했다. 이는 '#'로 시작하거나 '\0'으로 시작하는(빈 줄이라는 말임) 줄을 무시함으로써, 줄 전체 단위의 간략한 주석을 다는 것이 가능해진다.

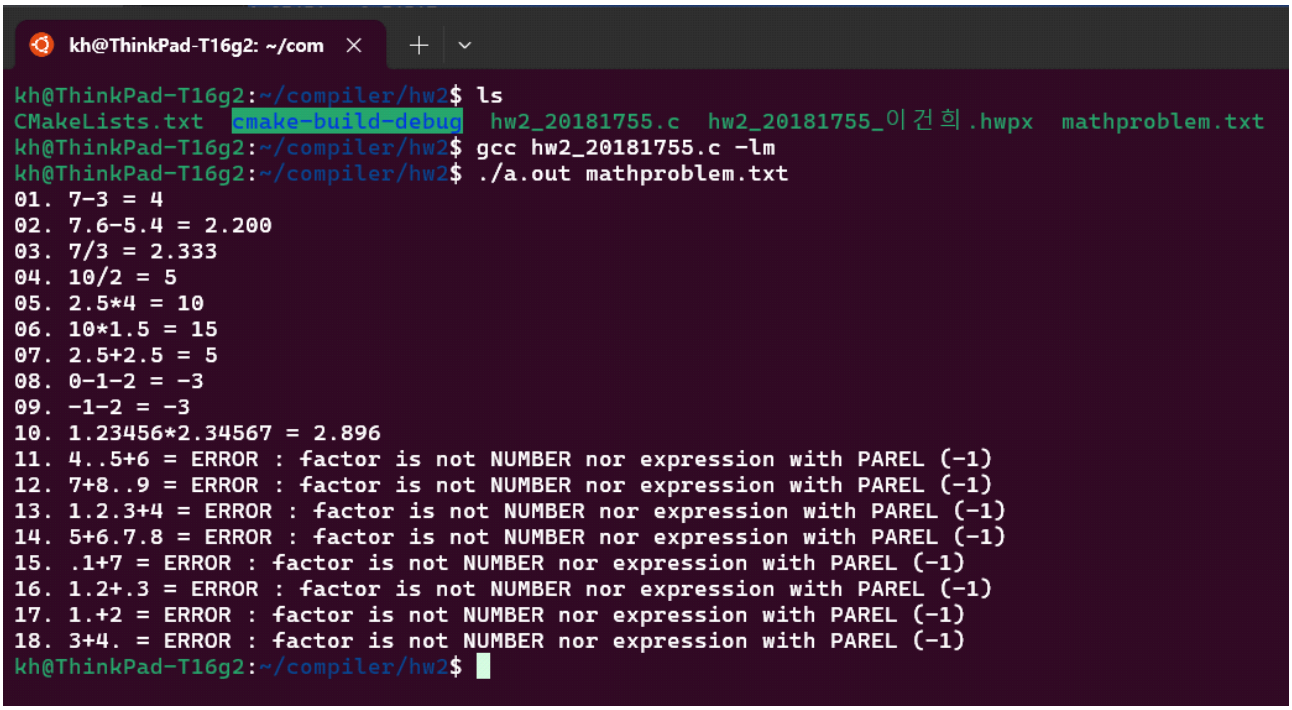
3. 실행 결과

본 과제는 1차 과제에서 기능을 덧붙인 것에 불과하다. 따라서 1차 과제에서도 언급한 세 가지의 테스트 유형;

1. parser가 지원하는 token에 대하여 올바른 수식
2. parser가 지원하는 token에 대하여 올바르지 않은 수식
3. parser가 지원하지 않는 token이 들어간 수식

중 3은 충분한 검증이 되었으니, 1과 2만 테스트를 진행하면 된다.

테스트 케이스 자체는 적으나 프로그램을 테스트 하기에 정말 의미 있는 수식만을 고민하고 엄선하여 넣었기에, 검증하기에는 충분하다.



테스트 파일은 부록 3에 있다. 자세한 테스트 내용은 가장 마지막 장을 참고 바란다.

4. 결론

수식 컴파일러도 이렇게 재밌는데, c언어 컴파일러 제작이 정말 기다려진다.

부록 1. 원시 프로그램 (with debug code)

** DEBUG 1을 0으로 두면 디버그 코드가 실행되지 않는다. **

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <stdbool.h>


/******************
 * 디버그 관련 *
 ******************/
#define DEBUG 1
#if DEBUG
int debugrec = 0;
char *tokword[] = {"NOTHING", "NUMBER", "PLUS", "MINUS", "STAR", "SLASH",
"LPAREL", "RPAREL", "END"};
#endif


/******************
 * 선언 & 정의 *
 ******************/
#define MAX_LINE 1000 // 최대 수식의 갯수, 임의로 정함
#define MAX_LINE_LEN 256 // 수식 최대의 길이, 임의로 정함
#define PRECISION 3 // float의 정밀도, 소숫점 n번째 자리까지, 임의로 정함

typedef enum {
    NOTHING,     // 공백, 그 외 기타
    NUMBER,      // 수 (정수 or 실수)
    PLUS,        // '+'
    MINUS,       // '-'
    STAR,        // '*'
    SLASH,       // '/'
    LPAREL,      // '('
    RPAREL,      // ')'
    END,         // EOF
} token_type; // enum에서 처음을 NULL로 두면 오류가 난다.

typedef enum {
    INT,
    FLOAT,
} num_type;

typedef union {
    int i;
    float f;
} intfloat;

typedef struct {
    num_type t;
    intfloat x;
    int e; /** calc()의 마지막에만 수정 */
} val;

const val VALDFLT = {INT, 0, 0};


/******************
 * 프로그램내에서 공유해야 하는 전역변수 *
 ******************/

// 1. 토큰
token_type token; // 토큰의 종류를 저장
token_type tokenprev; // 숫자 이외 토큰 중복 방지

// 2. 스트링
char *suborigin; // substring의 처음 시작 -> 조작된 '0'
char *subptr; // substring 을 가리키는 포인터

// 3. 숫자
intfloat num; // 파싱하여 나온 숫자 임시 저장
num_type numtype; // 파싱하여 나온 숫자의 종류
char numbuf[MAX_LINE_LEN]; // 여러자리 정수 계산시 임시 저장하는 버퍼
int numbufprog; // 상기 버퍼의 사용량

// 4. 에러
int errstatus; // 에러 상태, 0이 아니면 비정상
#define RETERR return VALDFLT
#define EXITi(errstatus) if(errstatus) RETERR


/******************
 * 함수 *
 ******************/

// 1. recursive-descent parser 핵심 함수

void yylex_number(); // 수를 파싱하는 코드가 너무 길어져서 따로 분리

void yylex();

val expression();

val term();

val factor();


// 2. 수 관련 함수 - 모든 수를 일단 float으로 다룬 후, 변환 가능하면 정수로 변환

void fconv(val lval, val rval, float *tmp1, float *tmp2);

float fcalc(float tmp1, float tmp2, token_type op);

val fcani(val r);


// 3. 편의를 위한 함수

val calc(char *line); // 강의교안 pseudo code의 main()

int countdigits(int number); // 10 degit 자릿수 계산

void printerr(int ecode); // 에러 출력


int main(int argc, char *argv[]) {
    /** 0. 프로그램 시작
     * 프로그램은 컴파일 이후 다음과 같이 작동하여야 한다
     * ./<program name> <math problems filename>
     */

    if (argc < 2) {
        printf("Error: Missing \"math problems file name\"\n");
        printf("Usage: %s <math problems file name>\n", argv[0]);
        exit(1);
    }


    /** 1. 파일 읽어오기
     * 파일에는 여러 줄의 수식이 있다.
     */

    FILE *fp = fopen(argv[1], "rt");
    if (fp == NULL) {
        printf("Error : math problem file does not exist\n");
        exit(1);
    }


    /** 2. 필요한 메모리 동적할당
     * 여러 수식을 계산할 것이기에, 필요한 메모리를 동적으로 할당해준다.
     */

    // 수식을 저장할 char 배열
    char **lines = calloc(sizeof(char *), MAX_LINE);

    for (int i = 0; i < 100; i++) {
        lines[i] = calloc(sizeof(char), MAX_LINE_LEN);
    }

    // 정답을 저장할 val 배열
    val *answers = calloc(sizeof(val), MAX_LINE);


    /** 3. 동적할당 한 메모리에 수식 저장 */
    int count = 0;

    while (1) {
        char buf[MAX_LINE_LEN];
```

```
        memset(buf, 0, sizeof(char) * MAX_LINE_LEN);
        buf[0] = '0';

        char *readline = fgets(buf + 1, MAX_LINE_LEN - 1, fp);
        buf[strcspn(buf, "\n\r")] = 0;
        if (readline == NULL) {
            break;
        }

        if(buf[1] == '#' || buf[1] == '\0'){
            continue;
        }

        int lidx = 0;
        for(int i=0; i<strlen(buf); i++){
            if(buf[i] != ' '){
                lines[count][lidx++] = buf[i];
            }
        }

        count++;
    }


    /** 4. 값 계산 */

    for (int i = 0; i < count; i++) {
        answers[i] = calc(lines[i]);

#if DEBUG
        printf("==============\n");
#endif

    }

    /** 5. 값 출력 */
    int digit = countdigits(count);

    float precision = powf(10, PRECISION);

    for (int i = 0; i < count; i++) {
        printf("%0*d. %s = ", digit, i + 1, lines[i] + 1);

        if (answers[i].e) { // 에러 발생했을 때
            printerr(answers[i].e);
        }
        else {
            if (answers[i].t == INT) {
                printf("%d\n", answers[i].x.i);
            }
            else {
                printf("%.*f\n", PRECISION,
                        (roundf(answers[i].x.f * precision) / precision));
            }
        }
    }


    /** 6. 동적할당 된 메모리 해제 */
    free(answers);

    for (int i = 0; i < MAX_LINE; i++) {
        free(lines[i]);
    }

    free(lines);


    return 0;
}


// 1. recursive-descent parser 핵심 함수

void yylex_number() {
    memset(numbuf, 0, MAX_LINE_LEN);
    numbufprog = 0;

    // 실질적인 숫자 읽기 시작
    numtype = INT;
    bool everdot = false;

    while (isdigit(subptr[0])) {
        numbuf[numbufprog++] = subptr[0];
        subptr++;

        if (subptr[0] == '.') {
            // 정수가 아닌 실수일 때

            // .의 이전이 조작된 0일 때
            if (subptr[-1] == '0' && subptr - 1 == suborigin) {
                token = NOTHING;
                errstatus = -1;
                return;
            }

            if (!everdot) {
                // 한 NUMBER에서 처음 나오는 '.'
                everdot = true;
                numtype = FLOAT;
                numbuf[numbufprog++] = subptr[0]; // '.'도 버퍼로 복사
                subptr++;

                if(!isdigit(subptr[0])){
                    token = NOTHING;
                    errstatus = -1;
                    return;
                }
            }
            else {
                // 한 NUMBER에서 두 번째 나오는 '.'
                token = NOTHING;
                errstatus = -1; // factor가 제대로 되지 않은 상황
                return;
            }
        }
    }

    if (numtype == INT) {
        num.i = (int) strtol(numbuf, NULL, 10);
    }
    else {
        num.f = strtof(numbuf, NULL);
    }
}

void yylex() {
// next token --> token
// number value --> num
    token = NOTHING;

#if DEBUG
    printf("%c, ", subptr[0]);
#endif

    /** 어떤 상황인지 판단하여 token 설정 */
    if (isdigit(subptr[0])) { // 수일 때
        token = NUMBER;
        yylex_number(); // subptr++은 yylex_number()의 마지막에서 이루어짐
    }
    else if (subptr[0] == '+') {
        token = PLUS;
        subptr++;
    }
    else if (subptr[0] == '-') {
        token = MINUS;
        subptr++;
    }
    else if (subptr[0] == '*') {
        token = STAR;
        subptr++;
    }
    else if (subptr[0] == '/') {
        token = SLASH;
        subptr++;
    }
    else if (subptr[0] == '(') {
        token = LPAREL;
        subptr++;
    }
    else if (subptr[0] == ')') {
        token = RPAREL;
        subptr++;
    }
```

```
    else if (subptr[0] == 0) {
        token = END;
        subptr++;
    }
    else {
        token = NOTHING;
        errstatus = -1;
        subptr++;
    }

    /**
     * 중복 가능 토큰 : (, )
     * 이외에는 전부 중복 불가능
     */
    if (!(token == LPAREL || token == RPAREL) && (token == tokenprev)) {
        token = NOTHING;
    }


#if DEBUG
    printf("(tokenprev, token) : (%s, %s)\n", tokword[tokenprev], tokword[token]);
#endif

    tokenprev = token;
}

val expression() {

#if DEBUG
    debugrec++;
    for (int i = 0; i < debugrec; i++) {
        printf("    ");
    }
    printf("** expression() **\n");
#endif

    val lval = VALDFLT;
    lval = term();
    token_type op = token;

    EXITi (errstatus);

    while (token == PLUS || token == MINUS) {
        yylex();
        val rval = term();

        float tmp1 = 0.0f;
        float tmp2 = 0.0f;

        fconv(lval, rval, &tmp1, &tmp2);

        val rslt = VALDFLT;
        rslt.t = FLOAT;
        rslt.x.f = fcalc(tmp1, tmp2, op);

        // 정수로 변환이 가능할 경우 변환해줌
        lval = fcani(rslt);
    }

#if DEBUG
    for (int i = 0; i < debugrec; i++) {
        printf("    ");
    }
    printf("** ~expression() **\n");
    debugrec--;
#endif

    return lval;
}

val term() {

#if DEBUG
    debugrec++;
    for (int i = 0; i < debugrec; i++) {
        printf("    ");
    }
    printf("** term() **\n");
#endif

    val lval = VALDFLT;
    lval = factor();
    token_type op = token;

    EXITi(errstatus);

    while (token == STAR || token == SLASH) {
        yylex();
        val rval = factor();

        float tmp1 = 0.0f;
        float tmp2 = 0.0f;

        fconv(lval, rval, &tmp1, &tmp2);

        /** dif by zero 에러 처리 */
        if (op == SLASH) {
            if ((rval.t == INT && rval.x.i == 0) ||
                (rval.t == FLOAT && rval.x.f == 0.0f)) {
                errstatus = -4;  // div by zero
                RETERR;
            }
        }

        val rslt = VALDFLT;
        rslt.t = FLOAT;
        rslt.x.f = fcalc(tmp1, tmp2, op);

        // 정수로 변환이 가능할 경우 변환해줌
        lval = fcani(rslt);

        op = token;
    }

#if DEBUG
    for (int i = 0; i < debugrec; i++) {
        printf("    ");
    }
    printf("** ~term() **\n");
    debugrec--;
#endif

    return lval;
}

val factor() {

#if DEBUG
    debugrec++;
    for (int i = 0; i < debugrec; i++) {
        printf("    ");
    }
    printf("** factor() **\n");
#endif

    val result = VALDFLT;

    EXITi(errstatus);

    if (token == NUMBER) {
        result.t = numtype;

        if (numtype == INT) {
            result.x.i = num.i;
        }
        else {
            result.x.f = num.f;
        }
        yylex();
    }

    else if (token == LPAREL) {
        yylex();
        result = expression();

        if (token == RPAREL) {
            yylex();
        }
        else {
            errstatus = -2;
            RETERR;
        }
    }
    else {
```

```
        errstatus = -1;
        RETERR;
    }

#if DEBUG
    for (int i = 0; i < debugrec; i++) {
        printf("      ");
    }
    printf("** ~factor() **\n");
    debugrec--;
#endif

    return result;
}


// 2. 수 관련 함수

void fconv(val lval, val rval, float *tmp1, float *tmp2) {
    if (lval.t == INT) {
        *tmp1 = (float) lval.x.i;
    }
    else {
        *tmp1 = lval.x.f;
    }

    if (rval.t == INT) {
        *tmp2 = (float) rval.x.i;
    }
    else {
        *tmp2 = rval.x.f;
    }

#if DEBUG
    printf("fconv() : %f %f\n", *tmp1, *tmp2);
#endif

}

float fcalc(float tmp1, float tmp2, token_type op) {
    switch (op) {
        case PLUS: {
            return tmp1 + tmp2;
        }
        case MINUS: {
            return tmp1 - tmp2;
        }
        case STAR: {
            return tmp1 * tmp2;
        }
        case SLASH: {
            return tmp1 / tmp2;
        }
        default: {
            return 0.0f;
        }
    }
}

val fcani(val r) {
    val ret = VALDFLT;

#if DEBUG
    printf("fcani1() : %f\n", r.x.f);
#endif

    if (fmod(r.x.f, 1.0) == 0.0) {
        ret.t = INT;
        ret.x.i = (int) r.x.f;
    }
    else {
        ret.t = FLOAT;
        ret.x.f = r.x.f;
    }

    ret.e = r.e;

#if DEBUG
    printf("fcani2() : %f\n", ret.x.f);
#endif

    return ret;
}


// 3. 편의를 위한 함수

val calc(char *line) {
    subptr = line; // 처음 시작시 string 전체를 substring으로 생각
    suborigin = line;

    val result = VALDFLT;
    errstatus = 0;
    tokenprev = NOTHING;

    yylex();
    result = expression();

    if (errstatus) {
        val errval = {-1, 0, errstatus};
        return errval;
    }

    if (token != END) {
        val errval = {-1, 0, -3};
        return errval;
    }
    else {
        return result;
    }
}

int countdigits(int number) {
    int digits = 0;

    while (number != 0) {
        number /= 10;
        digits++;
    }

    return digits;
}

void printerr(int ecode) {
    printf("ERROR : ");

    switch (ecode) {
        case -1: {
            printf("factor is not NUMBER nor expression with PAREL");
            break;
        }
        case -2: {
            printf("PAREL not closed");
            break;
        }
        case -3: {
            printf("given expression did not end properly");
            break;
        }
        case -4: {
            printf("div by zero");
            break;
        }
        default:
            printf("UNKNOWN CRITICAL ERROR");
    }

    printf(" (%d)\n", ecode);
}
```

부록 2. CMakeLists.txt

math.h를 사용했기에, 해당 library를 link해줘야 한다.
gcc로 컴파일 할 경우 파라메터에 -lm을 넣어주면 된다.

```
cmake_minimum_required(VERSION 3.0)

project(hw2_20181755)

set(CMAKE_C_STANDARD 99)

add_executable(hw2_20181755
        hw2_20181755.c
)

target_link_libraries(hw2_20181755 m)
```

부록 3. 테스트 파일 (mathproblem.txt)

```
# 추가 된 기능 + 2-5에서 자랑한 파일 읽어올 시 공백 삭제
7 -3
7.6 - 5.4
7/ 3

# 2-1에서 자랑했던 답이 정수일 시 정수로 변환하는 기능
10/2
2.5*4
10*1.5
2.5+2.5

# 2-2에서 자랑했던 MINUS로 시작하는 수식
# 아래 두 줄은 파서가 같은 방식으로 파싱
0-1-2
-1-2

# 2-3에서 자랑했던 정밀도
# define 된 PRECISION의 값을 바꿔보기도 해볼 것
1.23456 * 2.34567
# 답은 2.8958703552

# 2-4에서 자랑했던 소수점 오류처리

# 2-4-1
4..5+6
7+8..9
1.2.3+4
5+6.7.8

# 2-4-2
.1+7
1.2+.3

# 2-4-3
1.+2
3+4.
```